ECE540/489 Assignment 5 Project

Zhaohan (Daniel) Guo #997658583 ID:guozhaoh

April 15, 2013

1 Overview of source

I included the READMEs from the previous assignments as a reference.

main.cpp – this came from the original template, and I haven't modified the the contents

print.(h/c) – from printsimple

common.h – common stuff for my code

bitvecset.(h/cpp) – dynamic bit set based off of the given bitvector.(h/c) – written in more standard C++ using STL

instr.(h/cpp) – lists of instructions

block.(h/cpp) – basic blocks and edges

cfg.(h/cpp) – control flow graph

dom.(h/cpp) – dominance (either normal or post)

loop.(h/cpp) – natural loops

dfa.(h/cpp) – dataflow analysis engine

reachdefs.(h/cpp) – reaching definitions analysis and def-use and use-def chains

loop.(h/cpp) – finding natural loops and preheaders

licm.(h/cpp) – loop invariant code motion

constant.(h/cpp) – constant propagation and folding

dce.(h/cpp) – deadcode elimination

temp.(h/cpp) – turns pseudo registers into temporary registers where possible

copyprop.(h/cpp) – copy propagation

cse.(h/cpp) – common subexpression elimination

interpreter.(h/cpp) – compile time interpretation

doproc.cpp – the function that does the stuff per procedure

2 Overview of Optimizations

2.1 General

I went for simplicity of implementation over efficiency. My optimizations mostly only make small, incremental changes to the instructions, before all of the necessary structures for the analysis is rebuilt again. For example, before every optimization, I rebuild the CFG again from the instruction list; if an optimization needs def-use chains, I rebuild the def-use chain from scratch before running the optimization. This is to avoid keeping the structures consistent and up to date in an online fashion, which is more difficult.

As I was pressed for time, I also went for optimizations that had the best payoff in terms of optimized speed vs. implementation time. My optimizations aren't fully general and in many cases only partially implemented to have the most impact on the testcases.

2.2 Constant Propagation and Folding

This is in constant.(h/cpp). I use the def-use and use-def chains made from reaching definitions instead of creating a new dataflow analysis. I do both constant propagation and folding. This is a shallow optimization, which means I only propagate constants once - from completely constant registers to their immediate uses. This needs to be executed in a loop to keep propagating the constants. For constant folding, I support many instructions, but I left out a few because they weren't going to help get more speed on the testcases.

2.3 Deadcode Elimination

This is in dce.(h/cpp). This is where I implemented a simplified version of deadcode elimination. I just assume all control flow instructions are essential, instead of doing a postdominator type check.

2.4 Loop Invariant Code Motion

This is in licm.(h/cpp). This is almost implemented exactly as in the lecture slides. I relaxed one of the constraints that an instruction must dominate all the exits of the loop to also allow it to be moved when the instruction has no effect on anything outside the loop. This is a tradeoff because it's possible to make the program slower if certain branches in the loop aren't taken, but I felt like this made sense for our testcases. I also don't move memory instructions because I don't do alias analysis.

2.5 Copy Propagation

This is in copyprop.(h/cpp). The distinction between temporary registers and pseudo registers makes this a little less straightforward, but basically only pseudo registers can be propagated. This is really helpful because my other optimizations often result in copying around things, but not so much on its own.

2.6 Common Subexpression Elimination

This is in cse.(h/cpp). I only support a subset of expressions that I try to eliminate, mainly the ones that appear in the testcases. I also do a slightly different version of reaching evaluations. Instead of finding the earliest reaching evaluation, I find the closest. This is due to my dataflow framework only being able to work with bit vectors. To make up for it, running this optimization several times should propagate the common subexpression up earlier and earlier and probably achieve the same effect.

2.7 Temporary Register Conversion

This is in temp.(h/cpp). This is an artifact of the SimpleSUIF IR and my optimizations. Because my optimizations have to copy and move things around, they use pseudo registers all the time. But after many optimization passes, the copies may all be gone and it may be possible to use a temporary register in lieu of a pseudo register. That is exactly what this optimization does - it looks for pseudo registers that are defined and used only once in a basic block, and converts it to a temporary register.

2.8 Compile Time Interpretation

This is in interpreter.(h/cpp). This is just for testprog.c. Instead of doing more loop optimizations like loop fusion or partial loop unrolling, I decided that since many of the functions were simple, it was better to just interpret them at compile time. This way, the whole loop disappears and the function becomes trivial, maximizing speed. My interpreter is very simple, and can only handle signed integers, addition and multiplication, and at most one unknown variable. It would be possible to handle much more, such as floating point numbers, more unary and binary operations, even simple memory operations with simple alias analysis.

2.9 Environment Flags

ECE540_DISABLE_CONSTANT_OPTS – constant propagation and folding

ECE540_DISABLE_DCE – deadcode elimination

ECE540_DISABLE_TEMP – temporary register conversion

ECE540_DISABLE_LICM – loop invariant code motion

ECE540_DISABLE_COPY_PROP – copy propagation

ECE540_DISABLE_CSE – common subexpression elimination

ECE540_DISABLE_INTERPRETER – compile time interpretation

3 Other Notes

3.1 Warning: cvt_to_trees - node 641 used across expressions (promoted)

Sometimes I get this kind of warning when running s2c. This is caused by a definition of a temporary register and the usage of that register being separated by some specific instructions (such as an assignment to a pseudo register) that cause s2c to put them into separate expressions. I just let this one be and didn't try to handle it. To handle it properly, you would need to reorder instructions, which will involve finding dependencies and some kind of instruction scheduling.

3.2 Temporary Registers

I was really not clear initially on how temporary registers were supposed to be used and their limitations. One thing that tripped me up was that SimpleSUIF considers function calls to also delimit basic blocks, whereas for our assignments we treated them like ordinary instructions, so temporary registers could not be used across a call instruction.